Strong and Weak Scalability Studies for the 2-D Poisson Equation on the Taki 2018 Cluster

Carlos Barajas and Matthias K. Gobbert (gobbert@umbc.edu)

Department of Mathematics and Statistics, University of Maryland, Baltimore County

Technical Report HPCF–2019–1, hpcf.umbc.edu > Publications

Abstract

The new 2018 nodes in the cluster taki in the UMBC High Performance Computing Facility contain two 18-core Intel Skylake CPUs and 384 GB of memory per node, connected by an EDR (Enhanced Data Rate) InfiniBand interconnect. Parallel performance studies for the memory-bound test problem of the Poisson equation in two spatial dimensions yield several conclusions for the operation of the CPU cluster in taki. Strong scalability studies demonstrate excellent performance when using multiple nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Weak scalability studies confirm that best throughput is achieved by using all cores on a shared-memory node. Comparisons to results on the 2009 and 2013 nodes bring out that core-per-core performance of serial code improvements have stalled, but that node-per-node performance of parallel code continues to improve due to the larger number of cores available on a node. These observations compel the recommendations that serial code should use the 2009 and 2013 nodes of taki and parallel code is needed to take full advantage of the high-memory 2018 nodes. Comparisons between several compilers and several implementations of the MPI standard justify the choice of the Intel suite as the default on taki.

1 Introduction

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Started in 2008 by more than 20 researchers from ten academic departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. The facility is open to UMBC researchers at no charge. Researchers can contribute funding for long-term priority access. System administration is provided by the UMBC Division of Information Technology, and users have access to consulting support provided by dedicated full-time graduate assistants. See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

In 2017, the user community, represented by 51 researchers from 17 academic departments and research centers across UMBC, was successful for a third time to secure a grant from the National Science Foundation through its MRI program (grant no. OAC–1726023) for the extension and state-of-the-art update of HPCF. The HPCF Governance Committee ultimately decided in 2017–2018 to order a new cluster from Dell using the funds of this grant. The tests reported here use the new 2018 portion of the CPU cluster in taki. This portion of the CPU cluster consists of 42 compute nodes with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels, 140 W power), for a total of 36 cores per node, 384 GB memory (12×32 GB DDR4), and a 120 GB SSD drive. Figure 1.1 shows a schematic of one of the compute nodes, showing also the two Intel UPI connections between the CPUs and indicating that each CPU has 6 memory channels to a DDR4 memory of 32 GB. The nodes are connected by a network of four 36-port EDR (Enhanced Data Rate) InfiniBand switches (100 Gb/s bandwidth, 90 ns latency) to a central storage of more than 750 TB. See the system description at hpcf.umbc.edu for photos, schematics, and more detailed information, also on the other portions of the taki cluster.

This report uses the same test problem that has been used repeatedly to test cluster performance, including in 2018 [3] on taki for the first time and [2] on Stampede2, in 2014 [9] and 2015 [10] on maya, in 2010 [12, 13] on tara, in 2008 [6] on hpc, and in 2003 [1] on kali. The problem is the numerical solution of the Poisson equation with homogeneous Dirichlet boundary conditions on a unit square domain in two spatial dimensions. Discretizing the spatial derivatives by the finite difference method yields a system of linear equations with a large, sparse, highly structured, symmetric positive definite system matrix. This linear system is a classical test problem for iterative solvers and contained in several textbooks including [5, 7, 8, 14]. The parallel, matrix-free implementation of the conjugate gradient method as appropriate iterative linear solver for this linear system involves necessarily communications both collectively among all parallel processes and between pairs of processes in every iteration. Therefore, this method provides an excellent test problem for the overall, real-life performance of a parallel computer on a memory-bound algorithm. The results are not just applicable to the conjugate gradient method, which is important in its own right as a representative of the class of Krylov subspace methods, but to all memory-bound algorithms. The implementation uses the C programming language, with MPI (Message Passing Interface) for communications between distributed-memory cluster nodes and with OpenMP multi-threading on the shared-memory nodes.

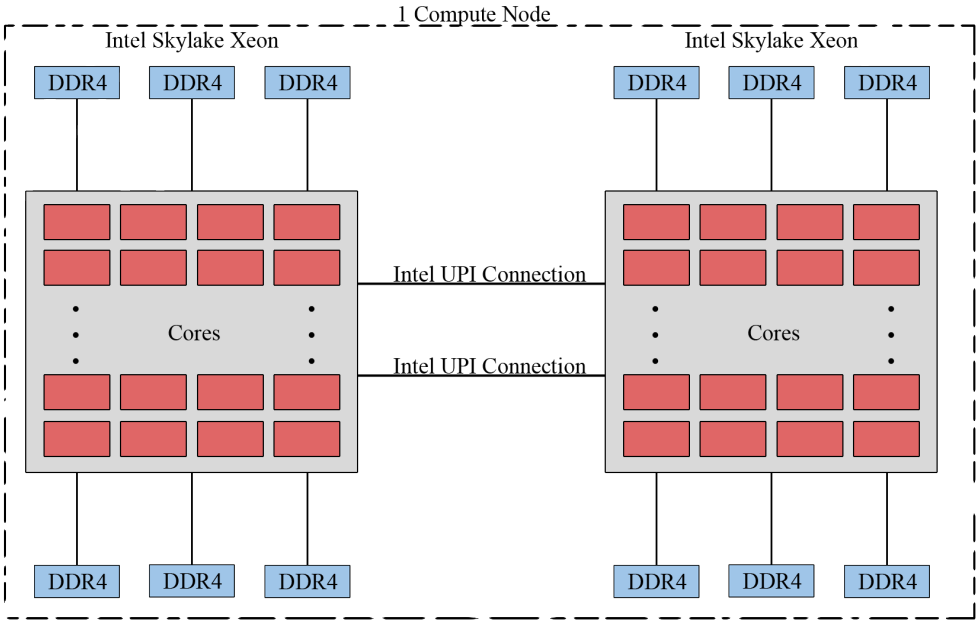


Figure 1.1: Schematic of a compute node with two Intel Skylake Xeon CPUs.

The studies in this report allow for a number of conclusions that underly decisions for the operation of the taki cluster in HPCF:

(1) With the pooled memory of 32 high-memory nodes of the 2018 portion of taki, a convergence study of the finite difference method is now possible up to a mesh size of $N \times N = 131072 \times 131072$ mesh points, requiring a system of linear equations with dimension $n = N^2 = 17{,}179{,}869{,}184$ or over 17 billion equations to be solved. The runtimes observed for serial jobs in this study motivates the use of parallel computing to significantly speed up the studies.

(2) The strong scalability studies demonstrate excellent performance when using multiple nodes due to the low latency of the high-performance interconnect and good speedup when using all cores of the multi-core CPUs. Comparisons to past results bring out that core-per-core performance of serial code improvements have stalled, but that node-per-node performance of parallel code continues to improve due to the larger number of cores available on a node. This justifies usage rules that direct serial runs with moderate memory needs to the 2009 and 2013 portions of the CPU cluster and parallel jobs or jobs with large memory needs to the 2018 portion of the CPU cluster. For jobs using large numbers of processes, studies of all combinations of numbers of nodes and numbers of processes per node show that best throughout is typically achieved by using the fewest number of nodes that can accommodate the desired numbers of processes. Moreover, weak scalability studies show that there is no downside to using MPI-only code also within a shared-memory node, whether only one node is used or multiple nodes.

(3) On taki, several compilers and several implementations of the MPI standard are installed. At this point in time, the following compilers are installed: the Intel compiler suite with icc as C compiler, the GNU compiler suite with gcc as C compiler, and the clang compiler. Each of these compilers has an OpenMP compile flag and is available in combination with the Intel implementation of MPI. Also the OpenMPI implementation of MPI is available on taki in combination with the Intel and the GNU compilers only, but performance has proven to be too slow to be viable at this point in time. performance has proven to be too slow to be viable at this point in time. The strong scalability studies for the combinations of icc, gcc, and clang with Intel MPI show that the combination of the Intel compiler and Intel MPI implementation is optimal by a small margin over the combinations with gcc and clang. This explains the choice of the Intel compiler suite and the Intel imeplementation of the MPI standard as defaults on taki.

The remainder of this report is organized as follows: Section 2 details the test problem and discusses the parallel implementation in more detail, and Section 3 summarizes the solution and method convergence data. Section 4 contains the strong scalability studies using MPI-only code on taki 2018 using the default Intel compiler and Intel MPI. Section 5 provides a historical comparison of performance of the results in this report and previous clusters in HPCF. Sections 6 and 7 contain the strong scalability studies using MPI-only code on taki 2018 using the GNU compiler and the clang compiler, respectively. Section 8 contains a weak scalability study using hybrid MPI+OpenMP code on taki 2018.

2 The Elliptic Test Problem

We consider the classical elliptic test problem of the Poisson equation with homogeneous Dirichlet boundary conditions (see, e.g., [14, Chapter 8])

$$\begin{aligned} -\triangle u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega, \end{aligned} \tag{2.1}$$

on the unit square domain $\Omega = (0,1) \times (0,1) \subset \mathbb{R}^2$. Here, $\partial\Omega$ denotes the boundary of the domain Ω and the Laplace operator in is defined as $\triangle u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}$. Using $N+2$ mesh points in each dimension, we construct a mesh with uniform mesh spacing $h = 1/(N+1)$. Specifically, define the mesh points $(x_{k_1}, x_{k_2}) \in \overline{\Omega} \subset \mathbb{R}^2$ with $x_{k_i} = h\, k_i$, $k_i = 0, 1, \ldots, N, N+1$, in each dimension $i = 1, 2$. Denote the approximations to the solution at the mesh points by $u_{k_1,k_2} \approx u(x_{k_1}, x_{k_2})$. Then approximate the second-order derivatives in the Laplace operator at the N^2 interior mesh points by

$$\frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_1^2} + \frac{\partial^2 u(x_{k_1}, x_{k_2})}{\partial x_2^2} \approx \frac{u_{k_1-1,k_2} - 2u_{k_1,k_2} + u_{k_1+1,k_2}}{h^2} + \frac{u_{k_1,k_2-1} - 2u_{k_1,k_2} + u_{k_1,k_2+1}}{h^2} \tag{2.2}$$

for $k_i = 1, \ldots, N$, $i = 1, 2$, for the approximations at the interior points. Using this approximation together with the homogeneous boundary conditions (2.1) gives a system of N^2 linear equations for the finite difference approximations at the N^2 interior mesh points.

Collecting the N^2 unknown approximations u_{k_1,k_2} in a vector $u \in \mathbb{R}^{N^2}$ using the natural ordering of the mesh points, we can state the problem as a system of linear equations in standard form $A\,u = b$ with a system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ and a right-hand side vector $b \in \mathbb{R}^{N^2}$. The components of the right-hand side vector b are given by the product of h^2 multiplied by right-hand side function evaluations $f(x_{k_1}, x_{k_2})$ at the interior mesh points using the same ordering as the one used for u_{k_1,k_2}. The system matrix $A \in \mathbb{R}^{N^2 \times N^2}$ can be defined recursively as block tri-diagonal matrix with $N \times N$ blocks of size $N \times N$ each. Concretely, we have

$$A = \begin{bmatrix} S & T & & & \\ T & S & T & & \\ & \ddots & \ddots & \ddots & \\ & & T & S & T \\ & & & T & S \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2} \tag{2.3}$$

with the tri-diagonal matrix $S = \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{N \times N}$ for the diagonal blocks of A and with $T = -I \in \mathbb{R}^{N \times N}$ denoting a negative identity matrix for the off-diagonal blocks of A.

For fine meshes with large N, iterative methods such as the conjugate gradient method are appropriate for solving this linear system. The system matrix A is known to be symmetric positive definite and thus the method is guaranteed to converge for this problem. In a careful implementation, the conjugate gradient method requires in each iteration exactly two inner products between vectors, three vector updates, and one matrix-vector product involving the system matrix A. In fact, this matrix-vector product is the only way, in which A enters into the algorithm. Therefore, a so-called matrix-free implementation of the conjugate gradient method is possible that avoids setting up any matrix, if one provides a function that computes as its output the product vector $q = A\,p$ component-wise directly from the components of the input vector p by using the explicit knowledge of the values and positions of the non-zero components of A, but without assembling A as a matrix.

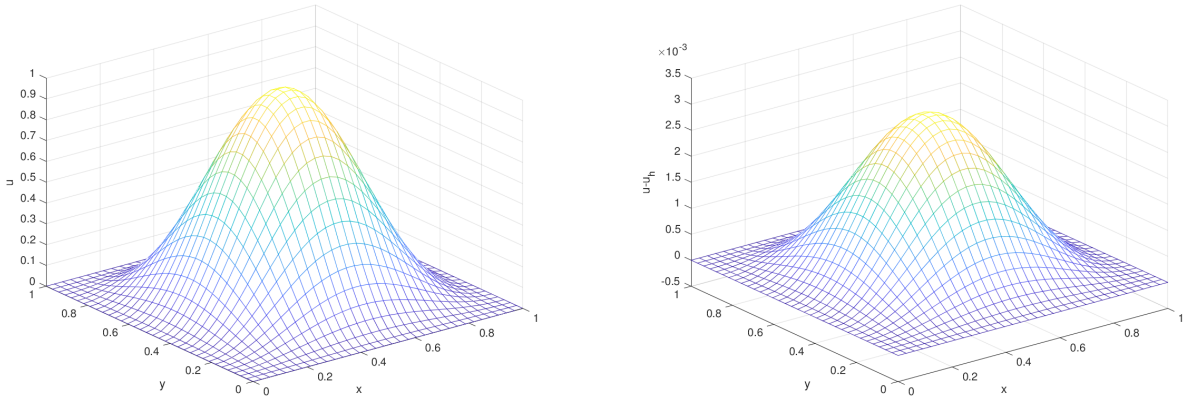
Thus, without storing A, a careful, efficient, matrix-free implementation of the (unpreconditioned) conjugate gradient method only requires the storage of four vectors (commonly denoted as the solution vector x, the residual r, the search direction p, and an auxiliary vector q). In a parallel implementation of the conjugate gradient method, each vector is split into as many blocks as parallel processes are available and one block distributed to each process. That is, each parallel process possesses its own block of each vector, and normally no vector is ever assembled in full on any process. To understand what this means for parallel programming and the performance of the method, note that an inner product between two vectors distributed in this way is computed by first forming the local inner products between the local blocks of the vectors and second summing all local inner products across all parallel processes to obtain the global inner product. This summation of values from all processes is known as a reduce operation in parallel programming, which requires a communication among all parallel processes. This communication is necessary as part of the numerical method used, and this necessity is responsible for the fact that for fixed problem sizes eventually for very large numbers of processes the time needed for communication — increasing with the number of processes — will

unavoidably dominate over the time used for the calculations that are done simultaneously in parallel — decreasing due to shorter local vectors for increasing number of processes. By contrast, the vector updates in each iteration can be executed simultaneously on all processes on their local blocks, because they do not require any parallel communications. However, this requires that the scalar factors that appear in the vector updates are available on all parallel processes. This is accomplished already as part of the computation of these factors by using a so-called Allreduce operation, that is, a reduce operation that also communicates the result to all processes. This is implemented in the MPI function `MPI_Allreduce` [11]. Finally, the matrix-vector product $q = A\,p$ also computes only the block of the vector q that is local to each process. But since the matrix A has non-zero off-diagonal elements, each local block needs values of p that are local to the two processes that hold the neighboring blocks of p. The communications between parallel processes thus needed are so-called point-to-point communications, because not all processes participate in each of them, but rather only specific pairs of processes that exchange data needed for their local calculations. Observe now that it is only a few components of q that require data from p that is not local to the process. Therefore, it is possible and potentially very efficient to proceed to calculate those components that can be computed from local data only, while the communications with the neighboring processes are taking place. This technique is known as interleaving calculations and communications and can be implemented using the non-blocking MPI communications commands `MPI_Isend` and `MPI_Irecv` [11]. For the hybrid MPI+OpenMP code, OpenMP threads are started on each MPI process using a `parallel for` pragma. This parallelizes each MPI process within the shared-memory of a node.

3 Convergence Study for the Model Problem

To test the numerical method and its implementation, we consider the elliptic problem (2.1) on the unit square $\Omega = (0,1) \times (0,1)$ with right-hand side function $f(x_1, x_2) = (-2\pi^2)\left(\cos(2\pi x_1)\,\sin^2(\pi x_2) + \sin^2(\pi x_1)\,\cos(2\pi x_2)\right)$, for which the true analytic solution in closed form $u(x_1, x_2) = \sin^2(\pi x_1)\,\sin^2(\pi x_2)$ is known. On a mesh with 32×32 interior points and mesh spacing $h = 1/33 \approx 0.030303$, the numerical solution $u_h(x_1, x_2)$ can be plotted vs. (x_1, x_2) as a mesh plot as in Figure 3.1 (a). The shape of the solution clearly agrees with the true solution $u(x_1, x_2)$ of the problem. At each mesh point, an error is incurred compared to the true solution $u(x_1, x_2)$. A mesh plot of the error $u - u_h$ vs. (x_1, x_2) is shown in Figure 3.1 (b). We see that the maximum error occurs at the center of the domain of size about 3×10^{-3} (note the scale on the vertical axis), which compares well to the order of magnitude $h^2 \approx 10^{-3}$ of the theoretically predicted error.

To check the convergence of the finite difference method as well as to analyze the performance of the conjugate gradient method, we solve the problem on a sequence of progressively finer meshes. The conjugate gradient method is started with a zero vector as initial guess and the solution is accepted as converged when the Euclidean vector norm of the residual is reduced to the fraction 10^{-6} of the initial residual. Table 3.1 lists the mesh resolution N of the $N \times N$ mesh, the number of degrees of freedom N^2 (DOF; i.e., the dimension of the linear system), the norm of the finite difference error $\|u - u_h\| \equiv \|u - u_h\|_{L^\infty(\Omega)}$, the ratio of consecutive errors $\|u - u_{2h}\| / \|u - u_h\|$, the number of conjugate gradient iterations `#iter`, the observed wall clock time in HH:MM:SS and in seconds, and the predicted



(a) Numerical solution u_h (b) Error $u - u_h$

Figure 3.1: Mesh plots of (a) the numerical solution u_h vs. (x_1, x_2) and (b) the error $u - u_h$ vs. (x_1, x_2).

Table 3.1: Convergence study of the finite difference method with serial code except where noted.

N	DOF	$\|u - u_h\|$	Ratio	`#iter`	wall clock time		memory usage (GB)	
					HH:MM:SS	seconds	predicted	observed
32	1,024	3.0128e–03	—	48	$<$ 00:00:01	$<$ 0.01	$<$ 1	$<$ 1
64	4,096	7.7811e–04	3.87	96	$<$ 00:00:01	$<$ 0.01	$<$ 1	$<$ 1
128	16,384	1.9765e–04	3.94	192	$<$ 00:00:01	0.01	$<$ 1	$<$ 1
256	65,536	4.9797e–05	3.97	387	$<$ 00:00:01	0.07	$<$ 1	$<$ 1
512	262,144	1.2494e–05	3.99	783	00:00:01	0.76	$<$ 1	$<$ 1
1024	1,048,576	3.1266e–06	4.00	1,581	00:00:08	7.76	$<$ 1	$<$ 1
2048	4,194,304	7.8019e–07	4.01	3,192	00:01:34	93.57	$<$ 1	$<$ 1
4096	16,777,216	1.9366e–07	4.03	6,452	00:13:38	817.55	$<$ 1	$<$ 1
8192	67,108,864	4.7392e–08	4.09	13,033	01:53:54	6,834.27	2	2.02
16384	268,435,456	1.1647e–08	4.07	26,316	15:28:12	55,692.09	8	8.02
32768	1,073,741,824	2.5840e–09	4.51	53,141	129:33:53	466,432.77	32	32.02
*65536	4,294,967,296	8.9964e–10	2.87	107,261	*02:39:58	*9,598.18	128	*148.11
*131072	17,179,869,184	3.0439e–10	2.96	216,433	*22:31:53	*81,112.96	512	*534.16

*This case uses 32 cores on 32 nodes; the observed memory is the total over all processes.

and observed memory usage in GB for studies performed in serial. More precisely, the serial runs use the parallel code run on one process only, on a dedicated node (no other processes running on the node), and with all parallel communication commands disabled by if-statements. The wall clock time is measured using the `MPI_Wtime` command (after synchronizing all processes by an `MPI_Barrier` command). The memory usage of the code is predicted by noting that there are $4N^2$ double-precision numbers needed to store the four vectors of significant length N^2 and that each double-precision number requires 8 bytes; dividing this result by 1024^3 converts its value to units of GB, as quoted in the table. The memory usage is observed in the code by checking the `VmRSS` field in the the special file `/proc/self/status`. The cases $N = 65536$ and 131072 require an excessive amount of runtime in serial as well as $N = 131072$ does not fit into the memory of one node. Therefore, 32 cores on 32 nodes are used for these cases, with observed memory summed across all running processes to get the total usage.

In nearly all cases, the norms of the finite difference errors in Table 3.1 decrease by a factor of about 4 each time that the mesh is refined by a factor 2. This confirms that the finite difference method is second-order convergent, as predicted by the numerical theory for the finite difference method [4, 8]. The fact that this convergence order is attained also confirms that the tolerance of the iterative linear solver is tight enough to ensure a sufficiently accurate solution of the linear system. For the two finest mesh resolutions, the reduction in the finite difference error is reduced, which points to the tolerance on the linear solver not being tight enough for these resolutions. The increasing numbers of iterations needed to achieve the convergence of the linear solver highlights the fundamental computational challenge with methods in the family of Krylov subspace methods, of which the conjugate gradient method is the most important example: Refinements of the mesh imply more mesh points, where the solution approximation needs to be found, and makes the computation of each iteration of the linear solver more expensive. Additionally, more of these more expensive iterations are required to achieve convergence to the desired tolerance for finer meshes. And it is not possible to relax the solver tolerance, because otherwise its solution would not be accurate enough and the norm of the finite difference error would not show a second-order convergence behavior, as required by its theory. For the cases up to $N \leq 32768$, the observed memory usage in units of GB rounds to within less than 1 GB of the predicted usage. This good agreement between predicted and observed memory usage in the last two columns of the table indicates that the implementation of the code does not have any unexpected memory usage in the serial case. For $N = 65536$ and 131072, the observed memory shows the memory usage totalled over all of the 1024 processes (32 cores on 32 nodes), which leads to a significant duplication of overhead, thus the observed memory usage is quite a bit larger than the predicted one. The wall clock times and the memory usages for these serial runs indicate for which mesh resolutions this elliptic test problem becomes challenging computationally. Notice that the very fine meshes show very significant runtimes and memory usage; parallel computing clearly offers opportunities to decrease runtimes as well as to decrease memory usage per process by spreading the problem over the parallel processes.

We note that the results in Table 3.1 agree with past results for this problem, see [3] and the references therein. This ensures that the parallel performance studies in the next section are practically relevant, since a correct solution of the test problem is computed.

4 Performance Studies on taki 2018 Using MPI-Only Code

This section presents the strong scalability studies using MPI-only code on taki 2018 using the default Intel compiler and Intel MPI. The Intel compiler `icc` and the Intel MPI implementation, currently version 18.0.3, are accessed on taki through the wrapper `mpiicc`. Since the compiler and MPI implementation are the defaults, they are available after the `module load default-environment` command in the `.bashrc` file in the user's home directory that is automatically executed upon login to taki. We use the compiler options `-O3 -std=c99 -Wall` along with the compiler's `march` settings `-xCORE-AVX2` and `-axCORE-AVX512` to generate an optimal binary for the Intel Skylake CPUs. We actually compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-qopenmp`, but set the environment variable `OMP_NUM_THREAD` to 1 at run time. This limits the MPI+OpenMP code to 1 thread per MPI process, which is equivalent to running MPI-only code compiled without OpenMP.

HPCF uses the slurm workload manager (slurm.schedmd.com) for job scheduling. The slurm submission script uses the `mpirun` command to start the job, with the option `-genv I_MPI_PIN_PROCESSOR_LIST allcores:map=scatter` that is supposed to optimize the placement of MPI processes for MPI-only / single-threaded jobs.[1] The number of nodes are controlled by the `--nodes` option and the number of MPI processes per node by the `--ntasks-per-node` option. For a performance study, each node that is used is dedicated to the job with the remaining cores idling by using the `--exclusive` flag. Correspondingly, we request all memory of the node for the job by `--mem=MaxMemPerNode`.

We include the `OMP_PLACES` and `OMP_PROC_BIND` environment variables[2] in the slurm script. The environment variable `OMP_PLACES=cores` is used to list the cores of the CPUs on the node as the places that OpenMP threads are pinned on, while `OMP_PROC_BIND` chooses the order of places in the pinning. The value `OMP_PROC_BIND=close` means that the assignment goes successively through the available places, while `OMP_PROC_BIND=spread` spreads the threads over the places. Alternatively, the setting `OMP_PROC_BIND=true` just prevents the operating system from moving threads around. Studies in [2] with `OMP_PLACES=cores` using the choices of `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread` resulted in runtimes that were almost the same for corresponding values of nodes and processes per node used, which should be expected, since the environment variables tested are supposed to influence the placement of OpenMP threads.

We conduct complete performance studies of the test problem for six progressively finer meshes of $N = 1024$, 2048, 4096, 8192, 16384, 32768; additionally, we demonstrate how 32 nodes can be leveraged to solve two yet finer meshes with $N = 65536$ and 131072. These studies result in progressively larger systems of linear equations with system dimensions ranging from about 1 million for $N = 1024$ to over 1 billion for $N = 32768$; the two final meshes with $N = 65536$ and 131072 have over 4 billion and over 17 billion unknowns, respectively. The results for the last two resolutions are contained in Table 3.1.

All possible nodes using all possible processes per node

Table 4.1 collects the results of the performance studies on the 2018 portion of the CPU cluster in taki. For each mesh resolution of the six meshes with $N = 1024$, 2048, 4096, 8192, 16384, 32768, the parallel implementation of the test problem is run on all possible combinations of nodes from 1 to 32 by powers of 2 and processes per node from 1 to 32 by powers of 2. The table summarizes the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format. The upper-left entry of each subtable contains the runtime for the 1-process run, i.e., the serial run, of the code for that particular mesh. The lower-right entry of each subtable lists the runtime using 32 cores on 32 nodes for a total of 1024 parallel processes working together to solve the problem. Notice that each node has two 18-core CPUs for a total of 36 cores, so even with 32 processes per node, several cores are not used by our job and remain available for the operating system and other system tasks.

We choose the mesh resolution of 16384×16384 in Table 4.1 to discuss in detail as example. Reading along the first column of this mesh subtable, we observe that by doubling the number of processes from 1 to 2 we approximately halve the runtime from each column to the next. We observe the same improvement from 2 to 4 processes as well as from 4 to 8 processes. We also observe that by doubling the number of processes from 8 to 16 processes, there is still a significant improvement in runtime, although not the halving we observed previously. Finally, while the decrease in runtime from 16 to 32 processes is small, the runtimes still do decrease, making the use of all available cores advisable. We observe that the behavior is analogous also in all other columns for this subtable. This behavior is a typical characteristic of memory-bound code such as this. The limiting factor in performance of memory-bound code is memory access, so we would expect a bottleneck when more processes on each CPU attempt to access the memory simultaneously than the available 6 memory channels per CPU indicated in Figure 1.1.

[1] Personal communication from Dell.

[2] http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html

Table 4.1: Wall clock time in HH:MM:SS using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:00:08	00:00:03	00:00:02	00:00:03	00:00:00	00:00:00
2 processes per node	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00
4 processes per node	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
8 processes per node	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
16 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
32 processes per node	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:01:34	00:00:43	00:00:18	00:00:07	00:00:03	00:00:02
2 processes per node	00:00:43	00:00:17	00:00:07	00:00:11	00:00:06	00:00:01
4 processes per node	00:00:22	00:00:09	00:00:04	00:00:02	00:00:01	00:00:00
8 processes per node	00:00:11	00:00:05	00:00:02	00:00:01	00:00:00	00:00:00
16 processes per node	00:00:08	00:00:02	00:00:01	00:00:01	00:00:00	00:00:00
32 processes per node	00:00:06	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:13:38	00:06:41	00:03:13	00:01:28	00:00:39	00:00:14
2 processes per node	00:06:42	00:03:12	00:01:29	00:00:37	00:00:14	00:00:07
4 processes per node	00:03:26	00:01:38	00:00:46	00:00:19	00:00:08	00:00:04
8 processes per node	00:01:46	00:00:51	00:00:24	00:00:10	00:00:04	00:00:02
16 processes per node	00:01:12	00:00:35	00:00:16	00:00:05	00:00:02	00:00:01
32 processes per node	00:01:03	00:00:30	00:00:13	00:00:03	00:00:01	00:00:01

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	01:53:54	00:56:50	00:28:14	00:13:48	00:06:37	00:03:04
2 processes per node	00:56:53	00:28:14	00:13:51	00:06:40	00:03:07	00:01:21
4 processes per node	00:29:13	00:14:23	00:07:05	00:03:26	00:01:38	00:00:41
8 processes per node	00:15:11	00:07:32	00:03:41	00:01:50	00:00:51	00:00:21
16 processes per node	00:10:10	00:05:03	00:02:29	00:01:13	00:00:36	00:00:12
32 processes per node	00:08:45	00:04:22	00:02:09	00:01:03	00:00:27	00:00:07

(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	15:28:12	07:41:17	03:47:46	01:54:36	00:56:34	00:27:46
2 processes per node	07:39:45	03:52:01	01:54:28	00:57:21	00:28:05	00:13:37
4 processes per node	03:56:37	01:58:27	00:58:52	00:29:23	00:14:29	00:07:04
8 processes per node	02:03:22	01:01:43	00:30:56	00:15:18	00:07:37	00:03:47
16 processes per node	01:22:16	00:41:19	00:20:39	00:10:18	00:05:07	00:02:30
32 processes per node	01:10:50	00:35:34	00:17:53	00:09:03	00:04:33	00:02:18

(f) Mesh resolution $N \times N = 32768 \times 32768$, system dimension 1073741824

	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	129:33:53	64:35:19	32:18:05	16:10:31	08:05:24	04:02:49
2 processes per node	64:35:55	32:18:39	16:17:10	08:07:41	04:04:48	02:02:33
4 processes per node	33:10:06	16:33:34	08:17:26	04:09:37	02:04:35	01:02:47
8 processes per node	17:21:21	08:40:33	04:20:30	02:10:04	01:05:15	00:32:58
16 processes per node	11:17:15	05:40:11	02:49:57	01:25:19	00:43:03	00:22:00
32 processes per node	09:38:57	04:50:32	02:26:02	01:13:41	00:37:21	00:19:21

Reading along each row of the 16384×16384 mesh subtable, we observe that by doubling the number of nodes used, and thus also doubling the number of parallel processes, we approximately halve the runtime all the way up to 32 nodes. This behavior observed for increasing the number of nodes confirms the quality of the high-performance InfiniBand interconnect. Also, we can see that the timings for anti-diagonals in Table 4.1 are about equal, that is for instance, the runtime for 2 nodes with 1 processes per node is almost same as for 1 node with 2 processes per node. Thus, it is advisable to use the smallest number of nodes with the largest number of processes per node.

When comparing now all subtables in Table 4.1, we observe that when we double the size of the mesh from one subtable to the next, the runtimes increase by a factor of about 8 to 10 for corresponding entries. The relative performance in each of the subtables in Table 4.1 exhibits largely analogous behavior to the 16384×16384 mesh, in particular the 8192×8192 and the 32768×32768 mesh subtables. For smaller meshes, some times for larger numbers of nodes are eventually so fast that improvement is small with more processes per node, but behavior is analogous for the more significant times.

All possible nodes using 32 processes per node

Parallel scalability is often visually represented by plots of observed speedup and efficiency. The ideal behavior of code for a fixed problem size N using p parallel processes is that it be p times as fast as serial code. If $T_p(N)$ denotes the wall clock time for a problem of a fixed size parameterized by N using p processes, then the quantity $S_p = T_1(N)/T_p(N)$ measures the speedup of the code from 1 to p processes, whose optimal value is $S_p = p$. The efficiency $E_p = S_p/p$ characterizes in relative terms how close a run with p parallel processes is to this optimal value, for which $E_p = 1$. The behavior described here for speedup for a fixed problem size is known as strong scalability of parallel code.

Table 4.2 organizes the results of Table 4.1 in the form of a strong scalability study, that is, there is one row for each problem size, with columns for increasing number of parallel processes p. Table 4.2 (a) lists the raw timing data, like Table 4.1, but organized by numbers of parallel processes p. Tables 4.2 (b) and (c) show the numbers for speedup and efficiency, respectively, that will be visualized in Figures 4.1 (a) and (b), respectively. There are several choices for most values of p, such as for instance for $p = 4$, one could use 1 node with 4 processes per node, 2 nodes with 2 processes per node, or 4 nodes with 1 process per node. In all cases, we use the smallest number of nodes possible, with 32 processes per node for $p \geq 32$; for $p < 32$, only one node is used, with the remaining cores idle. Comparing adjacent columns in the raw timing data in Table 4.2 (a) confirms our previous observation that performance improvement is very good from 1 to 2 processes, from 2 to 4 processes, and from 4 to 8 processes, while not quite as good from 8 to 16 processes and from 16 to 32 processes, but near-perfect halving of runtimes again for $p \geq 32$. The speedup numbers in Table 4.2 (b) help reach the same conclusions when speedup is essentially optimal with $S_p \approx p$ for $p \leq 8$. For $p = 16$ and 32, sub-optimal speedup is visible. The speedup numbers also indicate sub-optimal speedup for $p \geq 32$, but recall that the runtimes clearly showed halving from each column to the next one; the speedup numbers can only give this indication qualitatively. The efficiency data in Table 4.2 (c) can bring out these effects more quantitatively, namely efficiency is near-optimal $E_p \approx 1$ for $p \leq 8$, then clearly identifies the efficiency drop taking place for $p = 16$ and 32. But for $p \geq 32$, the efficiency numbers stay essentially constant, which confirms quantitatively the aforementioned halving of runtimes from each column to the next one for these columns.

The plots in Figures 4.1 (a) and (b) visualize the numbers in Tables 4.2 (b) and (c), respectively. These plots do not provide new results, but give a graphical representation of the data in Table 4.2. It is customary in strong scalability studies for fixed problem sizes that the speedup is better for larger problems, since the increased communication time for more parallel processes does not dominate over the calculation time as quickly as it does for small problems. This is born out generally by both plots in Figure 4.1. Specifically, the speedup plot in Figure 4.1 (a) for the two smallest meshes flattens quickly, while the lines for larger meshes continue to exhibit speedup. Even though the data are directly derived from the speedup data, the efficiency plot in Figure 4.1 (b) can provide additional insight into the observed behavior, particularly for small p. Here, the better-than-optimal behavior for $p \leq 32$ for the two smallest meshes is noticeable now. This excellent performance of runs on several processes can result from local problems that fit better into the cache of each processor, which leads to fewer cache misses and thus potentially dramatic improvement of the runtime, beyond merely distributing the calculations to more processes. For $p > 32$, we observe decreasing efficiency for the smallest meshes. For the larger meshes, the efficiency plot shows that the precipitous drop of efficiency takes place at $p = 16$ and 32; we already knew that from the data in Tables 4.2 (b) and (c), but the horizontal shape of the lines in the efficiency plot for $p > 32$ demonstrates readily that no further degradation of performance occurs, which shows the excellent behavior of the code for the largest meshes.

Table 4.2: Strong scalability study using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

(a) Wall clock time T_p in HH:MM:SS

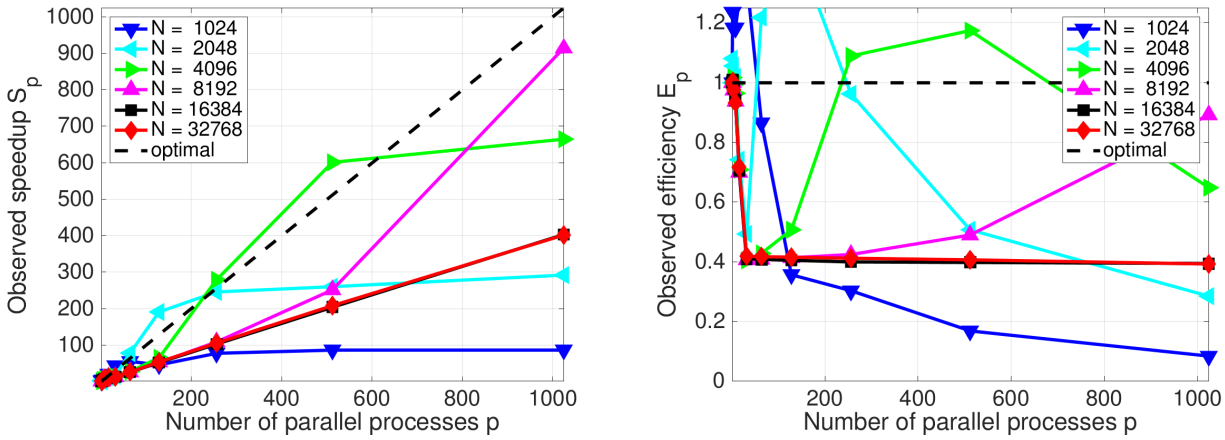
N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	00:00:08	00:00:03	00:00:02	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
2048	00:01:34	00:00:43	00:00:22	00:00:11	00:00:08	00:00:06	00:00:01	00:00:00	00:00:00	00:00:00	00:00:00
4096	00:13:38	00:06:42	00:03:26	00:01:46	00:01:12	00:01:03	00:00:30	00:00:13	00:00:03	00:00:01	00:00:01
8192	01:53:54	00:56:53	00:29:13	00:15:11	00:10:10	00:08:45	00:04:22	00:02:09	00:01:03	00:00:27	00:00:07
16384	15:28:12	07:39:45	03:56:37	02:03:22	01:22:16	01:10:50	00:35:34	00:17:53	00:09:03	00:04:33	00:02:18
32768	129:33:53	64:35:55	33:10:06	17:21:21	11:17:15	09:38:57	04:50:32	02:26:02	01:13:41	00:37:21	00:19:21

(b) Observed speedup $S_p = T_1/T_p$

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	1.00	2.47	4.73	9.46	20.42	43.11	55.43	45.65	77.60	86.22	86.22
2048	1.00	2.16	4.23	8.15	11.86	15.78	77.97	190.96	246.24	259.92	292.41
4096	1.00	2.03	3.97	7.72	11.34	12.97	27.38	64.99	279.03	601.14	664.67
8192	1.00	2.00	3.90	7.50	11.20	13.01	26.12	52.86	108.69	250.98	913.67
16384	1.00	2.02	3.92	7.52	11.28	13.10	26.10	51.91	102.49	204.09	403.01
32768	1.00	2.01	3.91	7.47	11.48	13.43	26.76	53.23	105.50	208.12	401.88

(c) Observed efficiency $E_p = S_p/p$

N	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=256$	$p=512$	$p=1024$
1024	1.00	1.24	1.18	1.18	1.28	1.35	0.87	0.36	0.30	0.17	0.08
2048	1.00	1.08	1.06	1.02	0.74	0.49	1.22	1.49	0.96	0.51	0.29
4096	1.00	1.02	0.99	0.96	0.71	0.41	0.43	0.51	1.09	1.17	0.65
8192	1.00	1.00	0.97	0.94	0.70	0.41	0.41	0.41	0.42	0.49	0.89
16384	1.00	1.01	0.98	0.94	0.71	0.41	0.41	0.41	0.40	0.40	0.39
32768	1.00	1.00	0.98	0.93	0.72	0.42	0.42	0.42	0.41	0.41	0.39



(a) Observed speedup S_p (b) Observed efficiency E_p

Figure 4.1: Strong scalability study using MPI-only code on taki 2018 using the Intel compiler and Intel MPI.

5 Comparison of Performance Studies on Different Clusters

This section contains a comparison of performance studies for the test problem obtained on different distributed-memory clusters that existed at UMBC over time. The oldest cluster kali from 2003 was the first cluster with a high-performance interconnect at UMBC, that is, it was the first to have a second network dedicated to MPI communications, in addition to the primary management network. These clusters, acquired over the years, give an impression what was the state-of-the-art CPU model and what constituted a high-performance interconnect at that time, on the concrete example of the test problem used in this report. Additionally, we include the cluster Stampede2 in the Texas Advanced Computing Center (TACC), a nationally funded supercomputer, which has the same generation CPU as taki 2018.

Table 5.1 reports results for the test problem on the historical mesh resolution of $N = 4096$, which was the largest resolution that could be solved on kali in 2003. The following list briefly describes each cluster in the table, see the cited references for detailed descriptions.

Cluster kali (2003) [1]: This cluster was a 33-node distributed-memory cluster with 32 compute nodes including a storage node (with extended memory of 4 GB) containing the 0.5 TB central storage, each with two (single-core) Intel Xeon processors (2.0 GHz clock speed) and 1 GB of memory per node, connected by a Myrinet interconnect, plus 1 combined user/management node. Note that for the case of all cores on 1 node, that is, for the case of both (single-core) CPUs used simultaneously, the performance was worse than for 1 CPU and hence the results were not recorded at the time. The cluster kali was funded by a SCREMS grant from the National Science Foundation and owned by the Department of Mathematics and Statistics at UMBC.

Cluster hpc (2008) [6]: The cluster hpc was a 35-node distributed-memory cluster with 33 compute nodes plus 1 development and 1 combined user/management node, each with two dual-core AMD Opteron processors and at least 13 GB of memory, connected by a DDR InfiniBand network and with an InfiniBand-accessible 14 TB parallel file system. The cluster hpc was the first cluster of HPCF, when HPCF was formed by seed funding from UMBC and contributions of faculty in 2008.

Cluster tara (2009) [12]: The cluster tara was an 86-node distributed-memory cluster with two quad-core Intel Nehalem X5550 processors (2.6 GHz clock speed, 8 MB L3 cache, 3 memory channels) and 24 GB memory per node, a QDR InfiniBand interconnect, and 160 TB central storage. This cluster was the first one principally funded by an MRI grant from the National Science Foundation to the user community of HPCF.

Clusters maya 2009, 2010, 2013 (2015) [10]: The cluster tara became part of the cluster maya as maya 2009, and its QDR InfiniBand network was extended to the 2013 portion of maya. The results for maya 2009 used the identical hardware as the results for tara, but new middleware and new versions of the compiler and MPI implementation from 2013. The cluster maya 2010 was a 168-node distributed-memory cluster with two quad-core Intel Nehalem X5560 processors (2.8 GHz clock speed, 8 MB L3 cache, 3 memory channels) and 24 GB memory per node, connected by a DDR InfiniBand interconnect. The cluster maya 2013 was a 72-node cluster with two eight-core Intel E5-2650v2 Ivy Bridge CPUs (2.6 GHz clock speed, 20 MB L3 cache, 4 memory channels) and 64 GB memory per node, connected by same QDR InfiniBand as maya 2009. The cluster maya had a central storage of more than 750 TB.

Cluster taki 2018: The 2018 portion of the CPU cluster in taki consists of 42 compute nodes with two 18-core Intel Xeon Gold 6140 Skylake CPUs (2.3 GHz clock speed, 24.75 MB L3 cache, 6 memory channels) and 384 GB memory per node, connected by an EDR InfiniBand interconnect.

Cluster Stampede2 (2018) [2]: The last row in the table contains results for the Skylake portion of Stampede2 with each compute node having two 24-core Intel Xeon Platinum 8160 Skylake CPUs (2.1 GHz clock speed, 33 MB L3 cache, 6 memory channels) and 192 GB memory per node, connected by an Omni-Path network.

Results on kali: On the cluster kali from 2003, we observe a factor of approximately 30 speedup by increasing the number of nodes from 1 to 32. However by using both cores on each node we only see a factor of approximately 25 speedup. We do not observe the expected 64 factor speedup, since both CPUs on the node share a bus connection to the memory, which leads to contention in essentially synchronized algorithms like Krylov subspace methods. Hence, it is actually faster to leave the second CPU idling rather than to use both [1].

Results on hpc: Note that there are four cores on each node of cluster hpc from 2008, compared to just two on the cluster kali, since the CPUs are dual-core. We observe approximately fourfold speedup that we would expect by running it on four cores rather than one. By running on 32 nodes with one core per node we observe the expected speedup of approximately 32; more in detail, the speedup is slightly better than optimal, which is explained by the smaller portions of the subdivided problem on each node fitting better into the cache of the processors. We see this for the first time here, but it is a typical effect in strong performance studies, in which a problem that already fits on one node is divided into smaller and smaller pieces as the number of nodes grows. Finally, by using all cores on 32 nodes

Table 5.1: Runtimes (speedup) for $N = 4096$ on the clusters kali, hpc, tara, maya, taki 2018, and Stampede2.

Cluster (year)	serial (1 core) time	1 node all cores time (speedup)	32 nodes 1 core per node time (speedup)	32 nodes all cores time (speedup)
kali (2003) [1]	02:00:49	N/A (N/A)	00:04:05 (29.59)	00:04:49 (25.08)
hpc (2008) [6]	01:51:29	00:32:37 (3.42)	00:03:23 (32.95)	00:01:28 (76.01)
tara (2009) [12]	00:31:16	00:06:39 (4.70)	00:01:05 (28.86)	00:00:09 (208.44)
maya 2009 (2015) [10]	00:17:05	00:05:55 (2.89)	00:00:35 (29.29)	00:00:08 (128.13)
maya 2010 (2015) [10]	00:17:00	00:05:48 (2.93)	00:00:34 (30.00)	00:00:06 (170.00)
maya 2013 (2015) [10]	00:12:26	00:02:44 (4.55)	00:00:15 (49.07)	00:00:12 (62.17)
taki 2018 (Section 4)	00:13:38	00:01:03 (12.98)	00:00:14 (58.43)	00:00:01 (818.00)
Stampede2 (2018) [2]	00:13:04	00:01:00 (13.07)	00:00:16 (49.00)	00:00:01 (784.00)

we observe a speedup of 76.01, less than the optimal speedup of 128, for a still respectable efficiency of 59.38% [6].

Results on tara: On the cluster tara from 2009, we observe a less than optimal speedup of approximately 5 by running on all 8 cores rather than on one, caused by the cores of a CPU competing for memory access. By running on 32 nodes with one core per node we observe a speedup of approximately 30. Finally, by using all 8 cores on 32 nodes we observe a speedup of 208, less than the optimal speedup of 256, but an excellent efficiency of 81.25% [12].

Results on maya: On maya 2009, we observe that by running on all 8 cores on a single node rather than 1 core there is a speedup of approximately 3 rather than the optimal speedup of 8. By running on 32 nodes with one core per node we observe a speedup of approximately 29 of the possible 32, which is very good. Finally, by using all 8 cores on 32 nodes we observe a speedup of 128, half of the optimal speedup of 256, or an efficiency of 50.00% [10]. On maya 2010, we observe that by running on all 8 cores on a single node rather than 1 core there is a speedup of approximately 3 rather than the optimal speedup of 8. By running on 32 nodes with one core per node we observe a speedup of approximately 30. When combining the use of all cores with the use of 32 nodes, we observe a speedup of 170, short of the optimal speedup of 256, or an efficiency of 66.41% [10]. On maya 2013, we observe that by running on all 16 cores on a single node rather than on one core there is a speedup of approximately 5 rather than the expected speedup of 16. We observe a greater than optimal speedup of 49.07 by running on 32 nodes with one process per node; this is caused by the relatively small problem fitting into cache after dividing it onto 32 nodes, together with the quality of the QDR InfiniBand interconnect. [10].

Results on taki 2018: Recall that each node in taki 2018 has two 18-core Intel Skylake CPUs, for a total of 36 cores available on each node. The data for "all cores" on 1 node and on 32 nodes uses actually 32 cores per node. We can observe by reading down the first column of Table 5.1 that in the past the core per core performance improved each time. But this is not true any more now, as serial code on a single core in the Skylake processor is not faster than in the Ivy Bridge CPUs from 2013; compare the clock speeds of each CPU specified above. This demonstrates that using multi-threading and/or MPI on a shared-memory node is absolutely vital to achieving optimal and scalable performance today and in the future. The possibility of this can be seen by the 13-fold speedup of the code that is possible when using "all cores" of 1 node, which is an efficiency of only 41%, but still yields a time that is more than twice as fast than using all cores of maya 2013. In fact, when comparing the "all cores" results of taki 2018 to the previous rows, we see that they are the fastest in each column and an improvement over all past results. We also see that the EDR InfiniBand on taki 2018 provides for excellent speedup, as the 32-node results have better than optimal speedup, both comparing the 1-core and the 32-core cases, namely 58.43 instead of nominal 32 for the 1-core comparison, for instance. Then, for the all core comparison, the speedup of 818.00 of the nominal 1024 for 32 nodes with 32 cores gives an efficiency of 80% compared to the serial run, but when compared to all cores on 1 node it is a speedup of 63.00, which is actually better than the nominal 32 improvement to be expected. We note that these better than optimal results typically result from the problem being divided into such small chunks on each core that they fit in cache there, but the multi-node performance is still a testament to the low latency of the network.

Results on Stampede2: The last row of the table shows the results for the Stampede2 cluster, which has the same generation Intel Skylake CPUs as taki 2018, but with 24 cores each instead of 18, for a total of 48 cores per node. Like for taki 2018, the data for "all cores" on 1 node and on 32 nodes uses actually 32 cores per node of the available 48 cores. The observations for taki 2018 also apply here in essentially the same way. It is notable that the serial run is actually slightly faster than on taki 2018, despite the nominally lower clock rate on Stampede2; see above. But the 1-core on 32 nodes result is not faster, possibly owing to a different network, and none of the differences between taki 2018 and Stampede2 are significant.

6 Performance Studies on taki 2018 Using the GNU Compiler

This section presents the strong scalability studies using MPI-only code on taki 2018 using the GNU compiler and Intel MPI. The GNU C compiler `gcc` and the Intel MPI implementation, currently version 7.0.3 and version 18.0.3, respectively, are accessed on taki through the wrapper `mpicc`, which Intel advertises as the native way to use Intel MPI with alternate compilers. `gcc` and Intel MPI are loaded by default by the `module load default-environment` command in the `.bashrc` file in the user's home directory that is automatically executed upon login to taki. We use the compiler options `-O3 -std=c99 -Wall` along with the compiler's `march` setting `-march=skylake-avx512`. to generate an optimal binary for the Intel Skylake CPUs. We actually compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-fopenmp`, but set the environment variable `OMP_NUM_THREAD` to 1 at run time. This limits the MPI+OpenMP code to 1 thread per MPI process, which is equivalent to running MPI-only code compiled without OpenMP.

Table 6.1: Wall clock time in HH:MM:SS using MPI-only code on taki 2018 using the GNU compiler and Intel MPI.

Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:15:05	00:07:26	00:03:35	00:01:41	00:00:44	00:00:19
2 processes per node	00:07:25	00:03:35	00:01:41	00:00:44	00:00:19	00:00:10
4 processes per node	00:03:49	00:01:51	00:00:53	00:00:24	00:00:11	00:00:06
8 processes per node	00:02:02	00:01:00	00:00:29	00:00:13	00:00:06	00:00:03
16 processes per node	00:01:19	00:00:39	00:00:19	00:00:08	00:00:04	00:00:02
32 processes per node	00:01:06	00:00:32	00:00:14	00:00:05	00:00:05	00:00:02

Table 6.1 shows the runtimes for the mesh with $N = 4096$ using the GNU compiler and Intel MPI. Comparing to Table 4.1 (c) for the same mesh, the GNU C compiler does not perform better than the default Intel compiler.

7 Performance Studies on taki 2018 Using the clang Compiler

This section presents the strong scalability studies using MPI-only code on taki 2018 using the clang compiler and Intel MPI. The `clang` compiler and the Intel MPI implementation, currently version 7.0.0 and version 18.0.3, respectively, are accessed on taki through the wrapper `mpicc`, which Intel advertises as the native way to use Intel MPI with alternate compilers. Since the Intel MPI implementation is loaded by default, it is available after the `module load default-environment` command in the `.bashrc` file in the user's home directory that is automatically executed upon login to taki. However, since `clang` is not loaded by default, it must be loaded using `module load Clang/7.0.0` at the command line prior to compiling. Then, to use `clang` for compiling in `mpicc`, the flag `-cc=clang` must be used with `mpicc`. We use the compiler options `-O3 -std=c99 -Wall` along with the compiler's `march` setting `-march=skylake-avx512`. to generate an optimal binary for the Intel Skylake CPUs. We actually compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-fopenmp`, but set the environment variable `OMP_NUM_THREAD` to 1 at run time. This limits the MPI+OpenMP code to 1 thread per MPI process, which is equivalent to running MPI-only code compiled without OpenMP.

Table 7.1: Wall clock time in HH:MM:SS using MPI-only code on taki 2018 using the clang compiler and Intel MPI.

Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216						
	1 node	2 nodes	4 nodes	8 nodes	16 nodes	32 nodes
1 process per node	00:14:38	00:07:12	00:03:27	00:01:36	00:00:40	00:00:18
2 processes per node	00:07:11	00:03:27	00:01:37	00:00:42	00:00:18	00:00:09
4 processes per node	00:03:43	00:01:48	00:00:51	00:00:22	00:00:10	00:00:05
8 processes per node	00:02:00	00:00:59	00:00:29	00:00:12	00:00:05	00:00:03
16 processes per node	00:01:18	00:00:39	00:00:19	00:00:07	00:00:03	00:00:02
32 processes per node	00:01:05	00:00:32	00:00:14	00:00:04	00:00:02	00:00:01

Table 7.1 shows the runtimes for the mesh with $N = 4096$ using the clang compiler and Intel MPI. Comparing to Table 4.1 (c) for the same mesh, the clang compiler does not perform better than the default Intel compiler.

8 Performance Studies on taki 2018 Using Hybrid MPI+OpenMP Code

This section presents a parallel performance study using hybrid MPI+OpenMP code on taki 2018. Parallel communication functions from MPI are needed to use multiple nodes in a distributed-memory cluster. But on each node with memory that is shared across all cores of the node, OpenMP multi-threading offers another way to parallelize code. More precisely, in hybrid MPI+OpenMP code, OpenMP multi-threading parallelizes each MPI process, thus we have the opportunity to access a chosen number of computational cores on a node by a combination of MPI processes and OpenMP threads per MPI process. Since the amount of computing power used on each node is fixed and only the way to access them is different from one run to another, one should expect comparable runtimes and the study is a weak scalability study with respect to each node. Since the amount of computing power used still increases with more nodes used, one should expect faster runtimes and the study is a strong scalability study with respect to all nodes used.

The Intel compiler `icc` and the Intel MPI implementation, currently version 18.0.3, are accessed on taki through the wrapper `mpiicc`. We use the compiler options `-O3 -std=c99 -Wall` along with the compiler's `march` settings `-xCORE-AVX2` and `-axCORE-AVX512` to generate an optimal binary for the Intel Skylake CPUs. We compile hybrid MPI+OpenMP code by including OpenMP multi-threading with the compiler option `-qopenmp`. This OpenMP multi-threading is implemented using `#pragma` lines for performance-critical for-loops and other parallel portions of the code. The environment variable `OMP_NUM_THREAD` controls the number of OpenMP threads per MPI process at run time. This is set in the slurm submission script after the number of nodes are controlled by the `--nodes` option and the number of MPI processes per node by the `--ntasks-per-node`. For a performance study, each node that is used is dedicated to the job with the remaining cores idling by using the `--exclusive` flag. Correspondingly, we request all memory of the node for the job by `--mem=MaxMemPerNode`. The slurm submission script uses the `mpirun` command to start the job. For runs with `OMP_NUM_THREAD` equal to 1, `mpirun` is called with the option `-genv I_MPI_PIN_PROCESSOR_LIST allcores:map=scatter` that is supposed to optimize the placement of MPI processes for MPI-only / single-threaded jobs, while for `OMP_NUM_THREAD` greater than 1, `mpirun` is called with the option `-genv I_MPI_PIN_DOMAIN omp` that is appropriate for running hybrid MPI+OpenMP codes.[3] We also use the environment variables `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` in the slurm script.

Table 8.1 collects the results of the performance studies on the 2018 portion of the CPU cluster in taki. For each mesh resolution of the six meshes with $N = 1024$, 2048, 4096, 8192, 16384, 32768, the parallel implementation of the test problem is run on $N_n = 1, 2, 4, \ldots, 32$ nodes. On each node, 32 computational cores are used, accessed by a combination of p_n MPI processes per node and t_p OpenMP threads per MPI process such that their product $p_n\, t_p = 32$ total threads per node. Thus, 32 hardware cores per node are in use in each entry of the table, and with respect to the processes and threads per node, this study constitutes a weak scalability study, since the runtimes should nominally be the same, independent of how the software threads access the same number of hardware cores. With respect to the increasing number of nodes used, the study is a strong scalability study, since the amount of resources used increases and the runtimes should nominally halve from column to column with the doubling of nodes used. The table summarizes the observed wall clock time (total time to execute the code) in HH:MM:SS (hours:minutes:seconds) format.

Each subtable in Table 8.1 shows results for one mesh with resolution $N \times N$. Reading along each column in each subtable shows approximately the same runtime for each entry, confirming the weak scalability of the hybrid MPI+OpenMP code with respect to the fixed 32 computational cores per node. Reading along each row in each subtable shows an approximate halving of the runtime as the number of nodes doubles from column to column, confirming the strong scalability of the code with respect to increasing number of nodes used. More subtly, the last row of each subtable exhibits a slightly better performance than the other rows in nearly all cases. This is the row of single-threaded runs in each subtable that agrees up to experimental variability with the corresponding last row of each subtable in Table 4.1. This indicates how efficient a pure MPI-only code accesses the cores of each node.

[3] Personal communication from Dell.

Table 8.1: Wall clock time in HH:MM:SS using hybrid MPI+OpenMP code on taki 2018 using the Intel compiler and Intel MPI. Each run uses N_n nodes, p_n MPI processes per node, and $t_p = 32/p_n$ OpenMP threads per MPI process.

(a) Mesh resolution $N \times N = 1024 \times 1024$, system dimension 1048576

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
$p_n = 2$	$t_p = 16$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
$p_n = 4$	$t_p = 8$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
$p_n = 8$	$t_p = 4$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
$p_n = 16$	$t_p = 2$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00
$p_n = 32$	$t_p = 1$	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

(b) Mesh resolution $N \times N = 2048 \times 2048$, system dimension 4194304

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	00:00:07	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
$p_n = 2$	$t_p = 16$	00:00:07	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
$p_n = 4$	$t_p = 8$	00:00:07	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
$p_n = 8$	$t_p = 4$	00:00:07	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
$p_n = 16$	$t_p = 2$	00:00:07	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00
$p_n = 32$	$t_p = 1$	00:00:06	00:00:01	00:00:01	00:00:00	00:00:00	00:00:00

(c) Mesh resolution $N \times N = 4096 \times 4096$, system dimension 16777216

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	00:01:12	00:00:35	00:00:15	00:00:04	00:00:01	00:00:01
$p_n = 2$	$t_p = 16$	00:01:12	00:00:40	00:00:15	00:00:03	00:00:01	00:00:01
$p_n = 4$	$t_p = 8$	00:01:11	00:00:34	00:00:14	00:00:03	00:00:01	00:00:01
$p_n = 8$	$t_p = 4$	00:01:10	00:00:34	00:00:14	00:00:03	00:00:01	00:00:01
$p_n = 16$	$t_p = 2$	00:01:11	00:00:34	00:00:14	00:00:03	00:00:01	00:00:01
$p_n = 32$	$t_p = 1$	00:01:03	00:00:30	00:00:13	00:00:03	00:00:01	00:00:01

(d) Mesh resolution $N \times N = 8192 \times 8192$, system dimension 67108864

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	00:09:53	00:04:56	00:02:26	00:01:13	00:00:33	00:00:09
$p_n = 2$	$t_p = 16$	00:09:52	00:04:57	00:02:27	00:01:11	00:00:32	00:00:08
$p_n = 4$	$t_p = 8$	00:09:53	00:04:53	00:02:26	00:01:11	00:00:31	00:00:08
$p_n = 8$	$t_p = 4$	00:09:50	00:04:53	00:02:25	00:01:11	00:00:31	00:00:08
$p_n = 16$	$t_p = 2$	00:09:49	00:04:56	00:02:25	00:01:10	00:00:30	00:00:08
$p_n = 32$	$t_p = 1$	00:08:48	00:04:24	00:02:14	00:01:04	00:00:28	00:00:09

(e) Mesh resolution $N \times N = 16384 \times 16384$, system dimension 268435456

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	01:19:28	00:40:04	00:20:10	00:10:12	00:05:06	00:02:36
$p_n = 2$	$t_p = 16$	01:19:30	00:39:56	00:20:09	00:10:04	00:05:04	00:02:52
$p_n = 4$	$t_p = 8$	01:19:36	00:39:56	00:20:01	00:10:02	00:05:46	00:02:29
$p_n = 8$	$t_p = 4$	01:19:36	00:39:57	00:20:00	00:10:05	00:05:00	00:02:28
$p_n = 16$	$t_p = 2$	01:19:30	00:40:24	00:20:10	00:09:57	00:05:01	00:02:27
$p_n = 32$	$t_p = 1$	01:10:54	00:35:35	00:17:54	00:09:00	00:04:31	00:02:16

(f) Mesh resolution $N \times N = 32768 \times 32768$, system dimension 1073741824

		$N_n = 1$	$N_n = 2$	$N_n = 4$	$N_n = 8$	$N_n = 16$	$N_n = 32$
$p_n = 1$	$t_p = 32$	10:51:23	05:25:29	02:45:40	01:22:30	00:41:55	00:45:01
$p_n = 2$	$t_p = 16$	10:48:49	05:24:51	02:43:49	01:22:16	00:41:33	00:44:29
$p_n = 4$	$t_p = 8$	10:48:55	05:31:06	02:43:20	01:22:11	00:41:22	00:44:20
$p_n = 8$	$t_p = 4$	10:48:52	05:25:04	02:44:38	01:22:09	00:41:23	00:44:23
$p_n = 16$	$t_p = 2$	10:49:23	05:26:18	02:43:02	01:23:12	00:41:18	00:43:59
$p_n = 32$	$t_p = 1$	09:38:56	04:50:23	02:26:04	01:13:41	00:37:24	00:40:08

Acknowledgments

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258, CNS–1228778, and OAC–1726023) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources. Co-author Carlos Barajas was supported by UMBC as HPCF RA.

References

[1] Kevin P. Allen. Efficient parallel computing for solving linear systems of equations. *UMBC Review: Journal of Undergraduate Research and Creative Works*, vol. 5, pp. 8–17, 2004.

[2] Kritesh Arora, Carlos Barajas, and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the Stampede2 cluster and comparison of networks. Technical Report HPCF–2018–10, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.

[3] Carlos Barajas and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the 2018 portion of the Taki cluster. Technical Report HPCF–2018–18, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2018.

[4] Dietrich Braess. *Finite Elements*. Cambridge University Press, third edition, 2007.

[5] James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[6] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report HPCF–2008–1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

[7] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, vol. 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[8] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, second edition, 2009.

[9] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on maya 2013. Technical Report HPCF–2014–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[10] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report HPCF–2015–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[11] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[12] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report HPCF–2010–2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[13] Hafez Tari and Matthias K. Gobbert. A comparative study of the parallel performance of the blocking and non-blocking MPI communication commands on an elliptic test problem on the cluster tara. Technical Report HPCF–2010–6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[14] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley, third edition, 2010.